ECE 448 / CS 440

Spring 2018

Assignment One: Search

Search: Project Report

Griffin A. Tucker, Michael Racine, Kaleb Henderson
gtucker2, racine2, kjhende2
Three Credits
Mark Hasegawa-Johnson

Introduction

For this project, we consider the task of implementing a series of general-purpose search algorithms and applying them to solving maze-style puzzles. In 1.1 we address the problem of finding the most optimal path from a given start state to some end state; depth-first search, breadth-first search, greedy-first search, and the A* search are implemented for solving this problem. In 1.2 we take the problem considered in 1.1 and expand it to consider cases in which, for a maze puzzle, the goal state is the state in which a set of points are all traversed; a variation of the A* algorithm written for 1.1 is used to find the best path through this set of points. Our implementations of these algorithms as well as file I/O and user interfacing are written in the Python language, specifically Python 3.

Summary of Task Distribution

The work put into our solution may be partitioned into six general categories: setup and file I/O, data structure implementation, search implementation, heuristic development, testing, and documentation. The work found itself distributed as follows:

User interfacing and file I/O. Kaleb Henderson
Primary data structure implementation. Michael Racine
Secondary data structure implementation. Griffin A. Tucker
Search implementation. Griffin A. Tucker, Michael Racine
A* heuristic development. Griffin A. Tucker, Kaleb Henderson
Testing and bug correction. Griffin A. Tucker, Michael Racine
Write up. Griffin A. Tucker, Michael Racine

Preliminary Work

Before implementing the searches which defined this project, some work needed to be done to ensure that use of the final modules by individuals outside the development team occurred with some ease and that given information was converted into data representations which would prove useful to us and our implementations.

User Interfacing

Our search implementations are wrapped by a main module for user interfacing. This module performs simple tasks such as polling the user for names of files to operate upon, for searches to perform on these files, and for names to be given to files in exportation. This module delegates which search function to call from the search module, as well as calls upon the createmaze and MazeTree modules, modules used for file I/O and for the definition of our primary data structure respectively.

File I/O and Primary Data Structure Implementation

The puzzles which we were asked to solve were given to us as text documents. To collect and store the information contained in these files, Kaleb Henderson wrote the createmaze module. The loadMaze function takes a filename and a pointer to a list, opens the file specified

by the filename, and iterates over the characters contained in this file, storing them in the list. The resulting list pointed to by the list pointer is two-dimensional and is indexed with (x,y) coordinates. This list is used throughout our search implementations to reference data in the maze, and is accessed in the print_maze function, a function of the createmaze module, to parse maze information for file output. The loadMaze function returns an mInfo object, an object containing the height and width of the maze, the (x,y) coordinates of start point of the maze, and all (x,y) coordinates of any goal points contained in the maze. This information is used periodically throughout our search implementations and is necessary in the creation of our primary data structure.

In solving maze problems, the state space may be defined as the set of accessible points within the maze (along with their respective conditions and the status of the goal condition). The MazeTree module, written by Michael Racine, represents this state space as a tree with 4 children per node. Each node contains a pointer to its children, one per cardinal direction unless the child would be a maze wall or would outside of maze bounds. In addition, each node records the character data it contains with respect to its (x,y) position in the aforementioned 2D list of maze text data, as well as its (x,y) integer values and its status as to whether or not it has been traversed. If traversed, a node will store the direction it was approached from as well. The maze tree is created upon call of the create_tree function; this function returns a 2D array of pointers to all nodes in the structure, each node being stored in its corresponding (x,y) maze coordinate. This allows the "root" of the tree, i.e. the start location of a search to be any location which we need it to be, an aspect of the data structure which proved useful in our expansion of the A* search for the second half of this project.

Part 1.1. Basic Pathfinding

For this project, our first task was to develop a series of search implementations which would find paths from a given start state to some end state within a maze puzzle. Depth-first search, breadth-first search, greedy-first search, and the A* search were implemented for solving this problem. As will be discussed, the optimality of the path generated by a search algorithm varies with the implementation used; some searches are more efficient than others.

Depth-First Search

Search implementations may differ in the structure of their frontier list, a list of nodes in the search tree to be traversed. For a depth-first search, the frontier list takes the form of a last-in-first-out (LIFO) stack. If the node on the top of the stack does not satisfy the goal condition, it is expanded and all of its children are pushed to the stack. The top of the stack is repeatedly evaluated until the goal state is met or until all nodes have been evaluated. In our implementation, the goal condition may be defined as the character stored in the node being currently evaluated having the value of '.'. If the node did not contain the value '.', it was expanded and its children were then evaluated based on the ordering of the frontier list. One thing to note: during expansion, children of a node were marked as being visited from the

cardinal direction in which the current node is located with respect to them. This allowed us to retrace the final path from goal to start once the goal condition was met. Our depth-first search implementation was written by Griffin A. Tucker. The stack implementation used is to be accredited to Brad Miller and David Ranum (“Problem Solving with Algorithms and Data Structures using Python”). Generated by our implementation, illustrated solutions to test mazes succeed the following lists of statistical results:

Medium maze. Path cost, 121 steps; nodes expanded, 239 nodes.

Big maze. Path cost, 437 steps; nodes expanded, 655 nodes.

Open maze. Path cost, 56 steps; nodes expanded, 370 nodes

.

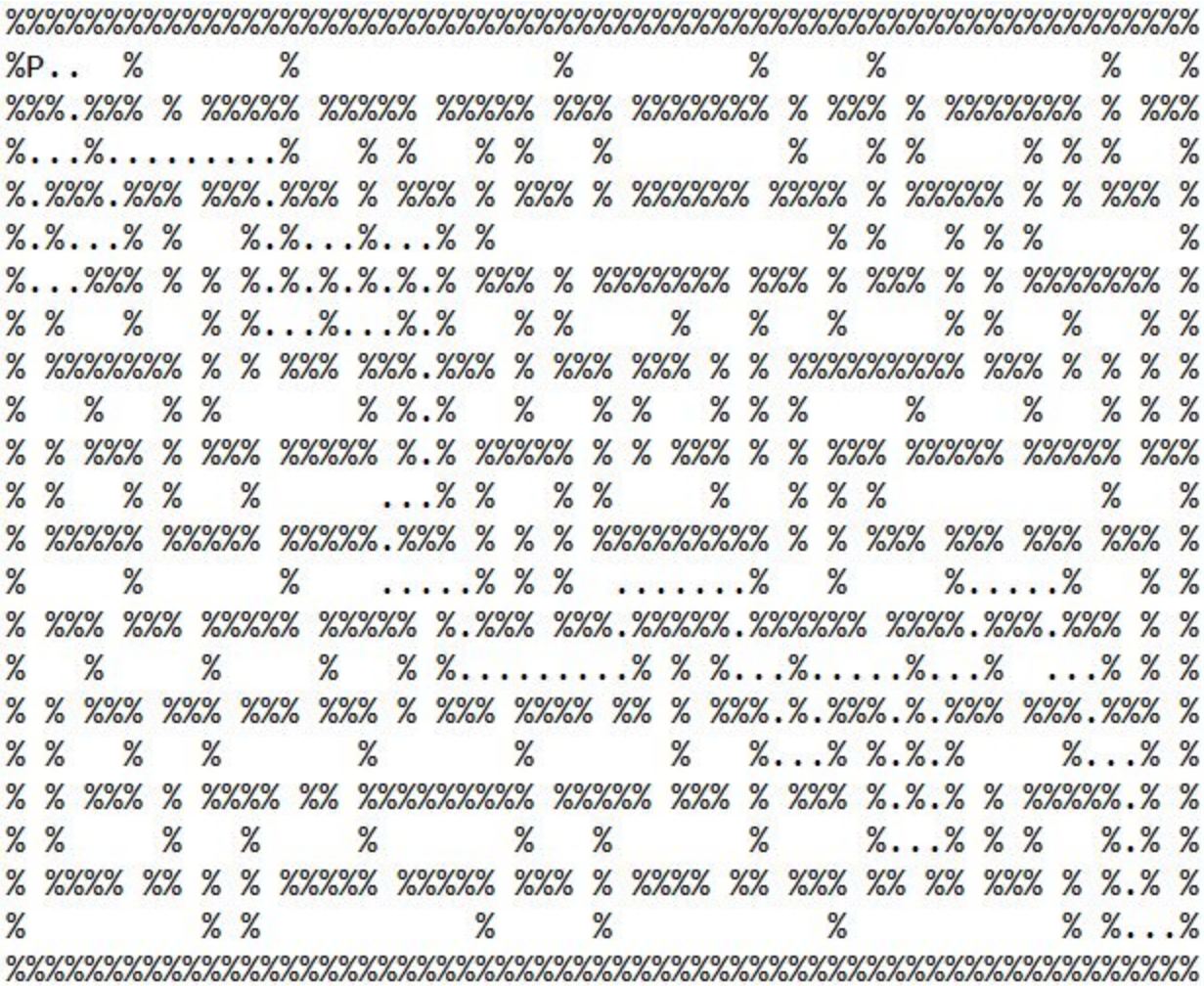

Figure 1. Medium Maze DFS Solution

Figure 2. Big Maze DFS Solution

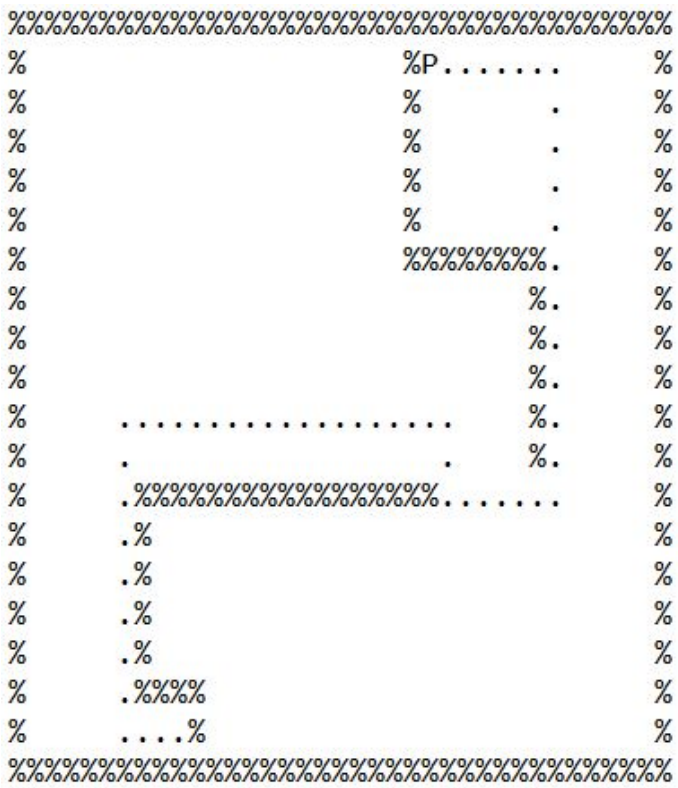

Figure 3. Open Maze DFS Solution

Breadth-First Search

A breadth-first search differs from a depth-first search in that its frontier list takes the form of a first-in-first-out (FIFO) queue. If the node at the front of the queue does not satisfy the goal condition, it is expanded and all of its children are pushed to the queue. The front of the queue is repeatedly evaluated until the goal state is met or until all nodes have been evaluated. In our implementation, the goal condition is the same as that in our implementation of depth-first search, i.e. the goal is met when the character stored in the node being currently evaluated has the value of '.'. If the node does not contain the value '.', it is expanded and its children are then evaluated based on the ordering of the frontier list. The same modifications to node children as are made in our depth-first search occur here as well. Our breadth-first search implementation was written by Griffin A. Tucker. The queue implementation used is to be accredited to Brad Miller and David Ranum ("Problem Solving with Algorithms and Data Structures using Python"). Generated by our implementation, illustrated solutions to test mazes succeed the following lists of statistical results:

Medium maze. Path cost, 105 steps; nodes expanded, 629 nodes.

Big maze. Path cost, 157 steps; nodes expanded, 1258 nodes.

Open maze. Path cost, 54 steps; nodes expanded, 565 nodes.

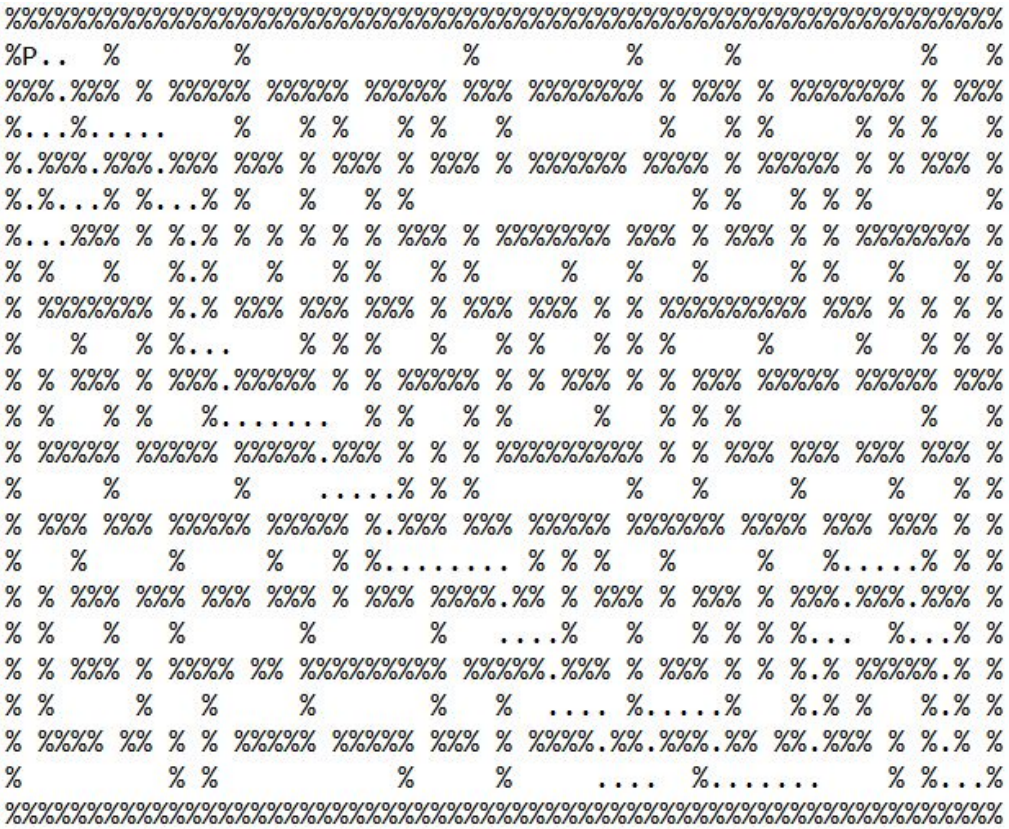

Figure 4. Medium Maze BFS Solution

Figure 5. Big Maze BFS Solution

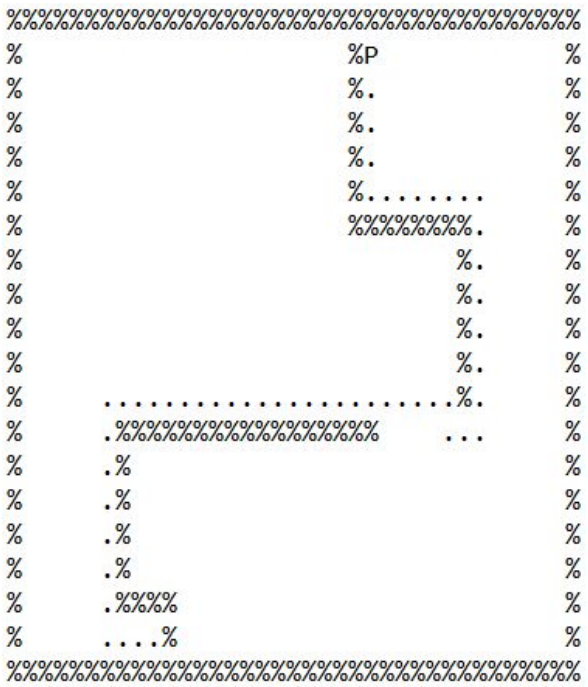

Figure 6. Open Maze BFS Solution

Greedy-First Search

Greedy-first search is distinguished from DFS and BFS through its use of an heuristic function; for our implementation, h(n) is the manhattan distance between n and the goal. Like BFS, the frontier list takes the form of a queue, however, the values on the queue are sorted by the heuristic function. Our greedy-first search takes the node from the front of the queue and expands it if it is not the goal. During expansion, it appends all of its neighbors to the queue in order of their heuristic value. The same modifications to node children as are made in our DFS and BFS occur here as well. Our greedy-first search implementation was written by Griffin A. Tucker. Generated by our implementation, illustrated solutions to test mazes succeed the following lists of statistical results:

Medium maze. Path cost, 104 steps; nodes expanded, 629 nodes.

Big maze. Path cost, 156 steps; nodes expanded, 1258 nodes.

Open maze. Path cost, 53 steps; nodes expanded, 565 nodes.

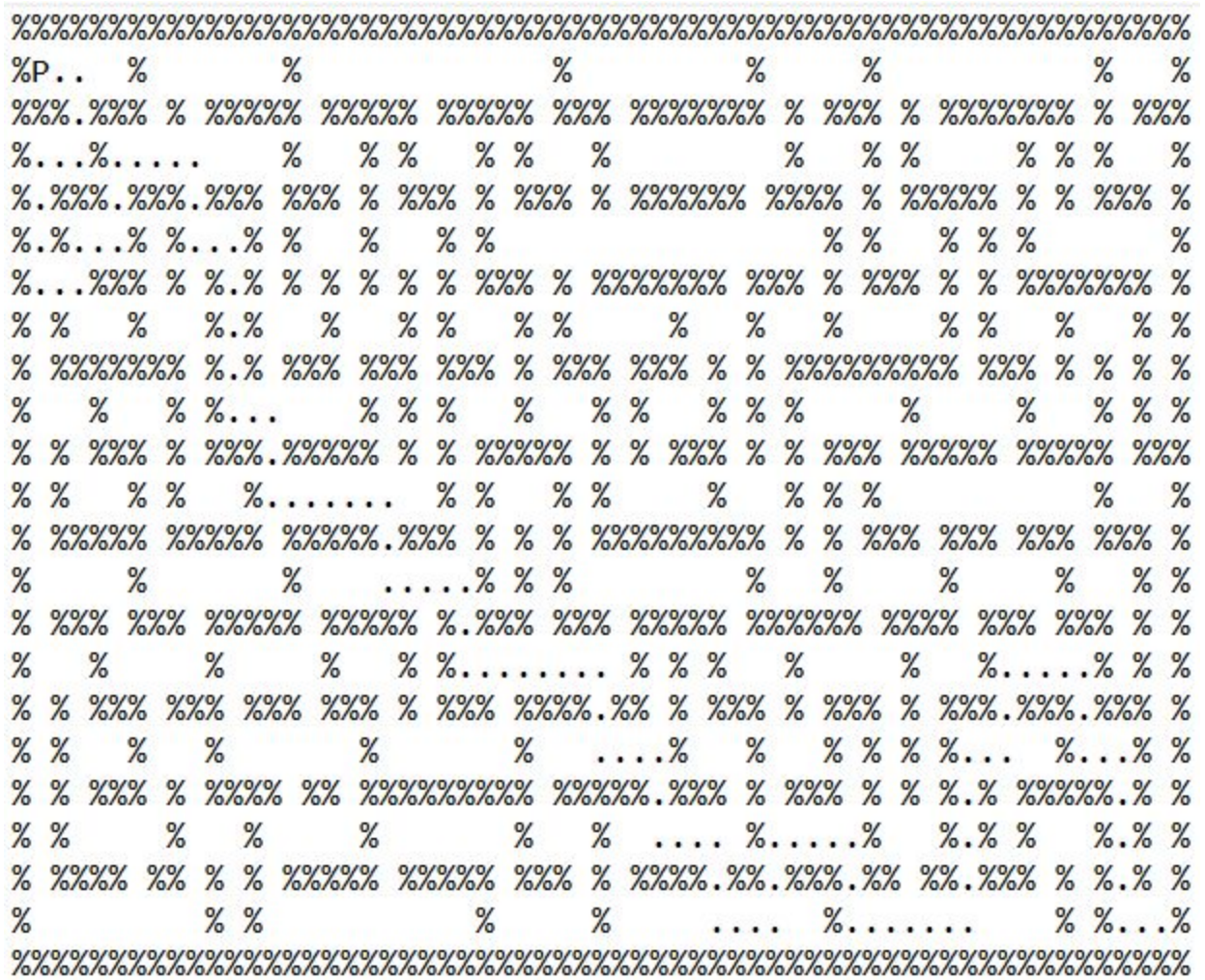

Figure 7. Medium Maze Greedy-First Solution

Figure 8. Big Maze Greedy-First Solution

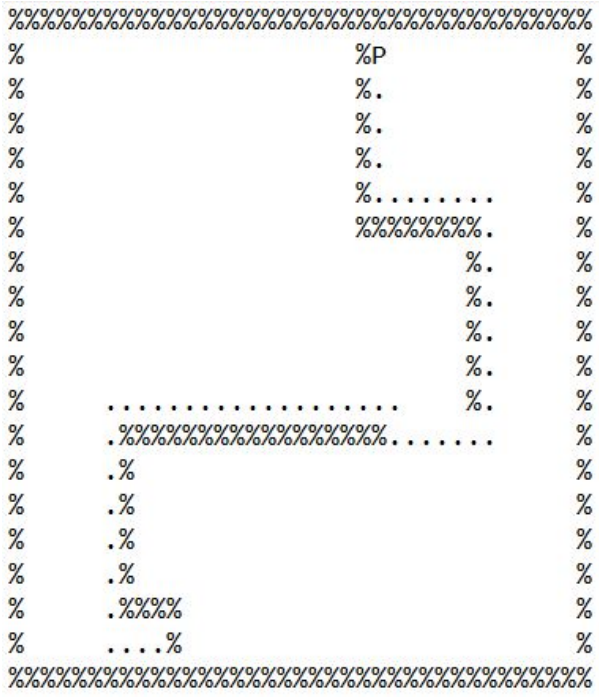

Figure 9: Open Maze Greedy First Solution

A* Search

Our A* implementation for one goal is similar to our greedy-first implementation in that it uses manhattan distance as its heuristic function; it differs in how it uses this function. Our implementation contains two dictionaries, f and g. The dictionary g maps the (x,y) coordinate of a node to the cost of going from that node to another location. The dictionary f maps the (x,y) coordinate of a node to the total cost of getting from the start location to the goal by passing through that node. g for some end node in an edge is the value of g for the start node of the edge plus one, one being the weight of the edge taken on in its traversal. f for some end node in an edge is the g value of the start node of the edge plus the manhattan distance of the end node to the goal. These values are used as follows: for every node in our frontier list, the minimum f value node is obtain and expanded if it is not the goal node. For every neighboring node during the expansion, their f and g value is calculated for future iterations. The algorithm iterates until the goal is found or until there are no more children to evaluate. Our A* search implementation was written by Griffin A. Tucker. Generated by our implementation, illustrated solutions to test mazes succeed the following lists of statistical results:

Medium maze. Path cost, 107 steps; nodes expanded, 529 nodes.

Big maze. Path cost, 159 steps; nodes expanded, 1338 nodes.

Open maze. Path cost, 54 steps; nodes expanded, 895 nodes.

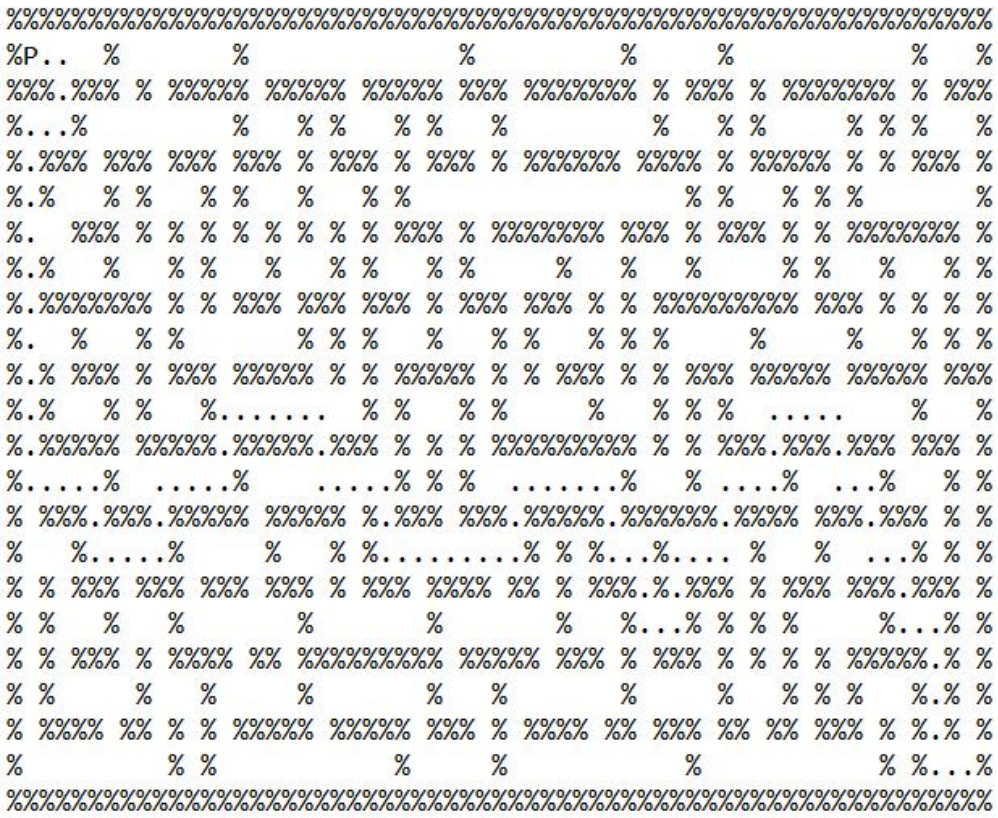

Figure 10. Medium Maze A* Solution

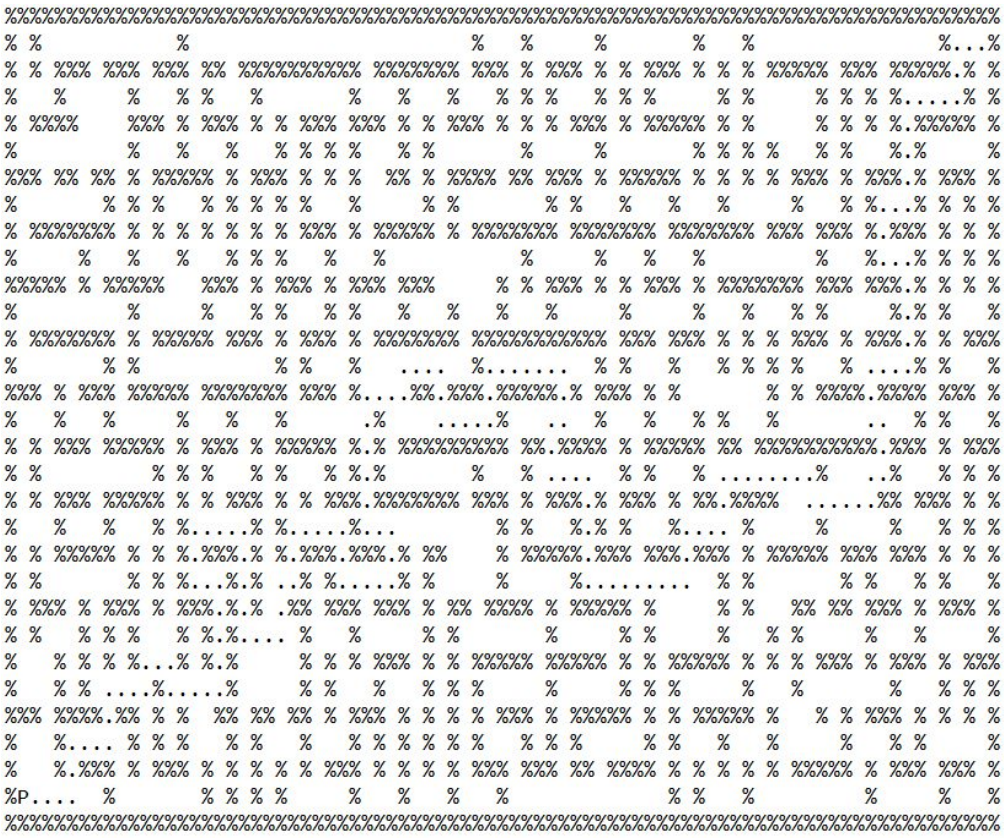

Figure 11. Big Maze A* Solution

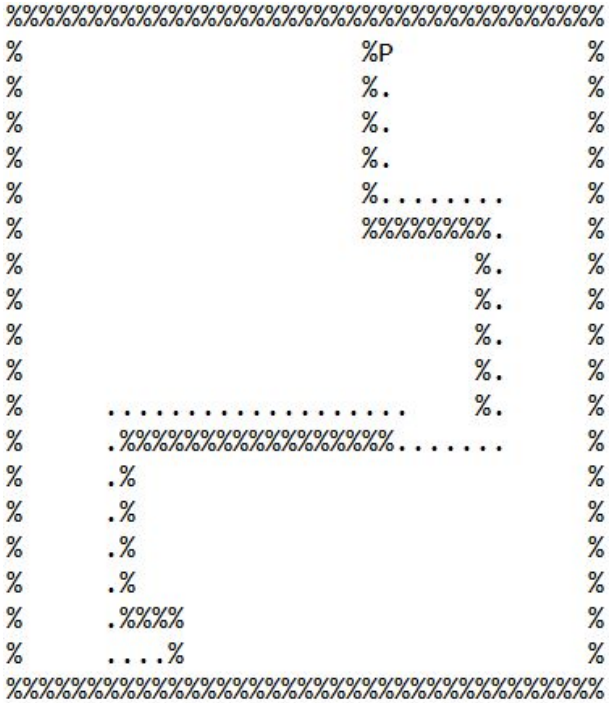

Figure 12. Open Maze A* Solution

1.1 Analysis

Observing the data returned by our search implementations, the following conclusions may be made: DFS returns the worst path for all mazes (as it always returns the first solution it finds), however, it expands the least amount of nodes; BFS returns an optimal solution and expands more nodes than DFS; greedy-first search is worse than BFS or is equivalent to it with the right heuristic (or our code has a bung which is making it operate the same); and A* finds an optimal solution expanding the least nodes out of BFS, Greedy, and itself. The following tables help organize the data used in making these conclusions:

	DFS	**BFS**	**Greedy**	**A***
Medium Maze	121	105	104	107
Big Maze	437	157	156	159
Open Maze	56	54	53	54

Table 1. Single-Goal Search Implementation; Step Cost

	DFS	**BFS**	**Greedy**	**A***
Medium Maze	239	629	629	529
Big Maze	655	1258	1258	1338
Open Maze	370	565	565	895

Table 1. Single-Goal Search Implementation; Nodes Expanded

Part 1.2. Search With Multiple Dots

With the proper modifications, our A* implementation from the previous section may be made able to search out the optimal path between not just one, but multiple goal points in a maze. The modifications we made include the implementation of two secondary data structures, done so in order to simplify the state space of the problem, and the development of a new heuristic, made to consider the specific constraints and requirements of a multi-goal maze puzzle.

State Space Simplification Through Secondary Data Structure Implementations

If we were to use the same state space as was in 1.1, our A* algorithm would run unnecessarily long for a multi-goal maze search. The number of possible transitions to consider would be numerous. By considering the paths between the goals themselves as opposed to the connections between the positions of the maze, we can greatly simplify the state space. In terms of data structures, this simplification would take a tree of nodes where each node is a position in

the 2D maze and each edge weight is equivalent to one, i.e. the distance between one node and its neighbor, and reduce it to a graph of nodes where each node is one goal point in the maze and each edge weight is equivalent to the path cost of an A* search between it and another goal point, i.e. the optimal path cost. There is an edge which represents the optimal path between every goal point in this graph. Note: the starting point is considered a goal point in this representation, as it must be traversed. So that it may be useful, this graph could then be converted into a minimum spanning tree, a tree whose connections make up the paths which must be taken in order to minimize the path cost of the traversal between every node in the preexisting graph. For this project, this concept was devised by Kaleb Henderson.

To implement the state space simplification, two data structures were defined within the sd_dict module: sd_dict and dict_mst. These structures were implemented by Griffin A. Tucker. The sd_dict structure contains a single Python dictionary which is filled upon instantiation with the optimal step distances between every goal point in the maze. Keys to this dictionary are tuples of (x,y) coordinates, i.e. a tuple where the each set of coordinates, tuples themselves, are the (x,y) integer coordinates two goal points in the maze. This dictionary is kept from having repeated edges in order to simplify future calculations, e.g. the tuple ((xA, yA),(xB, yB)) will never exist as a key at the same time as the tuple ((xB, yB),(xA, yA)) exists as a key. Each key is paired with the value of the optimal path cost between the two (x,y) points. These values are calculated by a series of A* searches performed at the instantiation of an sd_dict object. In terms of the prior discussed concept of this problem's simplification of state space, this object is the graph of all goal points in the maze. This object is used by the second data structure, dict_mst, to generate a minimum spanning tree for a maze using Kruskal's algorithm. The return value of dict_mst's function, create_mst, is similar to that which is returned by MazeTree, the primary data structure described in part 1.1; an array of nodes containing relevant information and pointers to their children which may be accessed with (x,y) coordinates. The relevant information stored in each node is the same as that which is stored in the nodes of a MazeTree object.

Heuristic Development

The information stored in dict_mst is essential for implementing our developed heuristic for the second half of the project. Considering traversed and non-traversed nodes in our A* algorithm, our heuristic function returns the sum of the minimum estimated distance to traverse all unvisited nodes in dict_mst (obtained from the module's sum_weights function) and the distance to the closest goal node with respect to the current node being evaluated. This heuristic is admissible, as for every node n, $h(n) <= h^*(n)$ where $h^*(n)$ is the true cost to reach the goal state from n. $h^*(n)$ is the sum of the weights of the edges between the remaining untraversed nodes of the mst and the weights of the edges which must be retraced to reach those edges. h(n) is always less than or equal to this value as it has no means of considering all required edge retraces; it only concerns itself with the remaining paths of the mst which remain untraced and

the path from one node to the nearest goal, which may or may not be a retrace. This heuristic was developed by Griffin A. Tucker.

Multi-Goal A* Search General Implementation

Our heuristic and secondary data structures required us to create a second variation of our A* implementation. This implementation, similar to our previous A* algorithm in structure, kept an additional list of the remaining unvisited nodes at every step of the traversal; when the length of the list reached 0, the search retraced the path and exported the solution. The function used for this implementation also kept a list of edges traversed, used during the retrace to export the correct path to a readable file. Our breadth-first search implementation was written by Griffin A. Tucker. Also, it should be noted that while the path generated is correct, being the path represented by the MST of the maze, the order in which the goal nodes were reached is incorrectly generated by our implementation. In addition, we have reason to believe that the step cost returned is incorrect as well. This is possibly because the nodes of the MST are being visited in an optimal order, leading some traversals from one node to another to span the size of the maze. These inaccuracies may be seen in the following illustrated solutions as generated by our implementation succeeding the following lists of statistical results:

Tiny search. Path cost, 72 steps; nodes expanded, 90 nodes.

Small search. Path cost, 244 steps; nodes expanded, 594 nodes.

Medium search. Path cost, 402 steps; nodes expanded, 912 nodes.

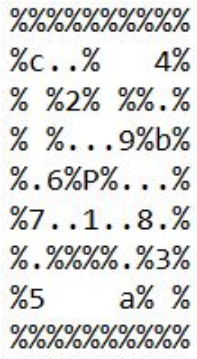

Figure 13. Tiny Search A* Solution

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%2.....P.1     %......4..8...%
%   %%%%%.%%%%%%.%. %.%.%%. .%
%   %  % .   %...%. %.%..%..e%
%    .d% ....b...%9...%..%%%%%
%%%%%.%%%%.%%%..%%%%%%%..... %
%a...............3    %.%%%. %
%%.%%%%%%%% %%%%%%%%%%%.%7.. %
%..     %        .......%.%%%%
%..     %%%%%.....%6......   %
%c% %%%    % .%   %% %%.%%%%%%
%          % f%        5     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 14. Small Search A* Solution

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%.c..%b...     %...........  %..f%     % %   %  %
%.. .%%%%.%%%%%%.%. % % %%.........%%%%  %2 %%% %
%.. .%9%....%.a%.%. % %  %  .%   %......... %   %
%...j............%d.....%%%%.%   %%.h%%%.%      %
%.%%%%%.%%%%%%  %%%%.%...........1% ...%3 %%%%%%%
%8..   .%...6%      .e% %%%. %%%%%% .%.%. %...% %
%%%.%% .%.%%%%%%% %%%%% %... %  5.%%.%.%....%4% %
%... % .%......%  %      P% %%  .....%...%%%%.% %
%7......%...% .%  %     %  ........%.......g%...%
% % %%%...i% %....%% %% %%%.%% %  %k%%%%%%%%%%%.%
%   %    %    %  ........... %    %.............%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 15. Medium Search A* Solution

1.2 Analysis

As aforementioned, some issues are present in our implementation of the multi-goal A* search algorithm. This can be seen by observing the data returned by the function. At first glance once can conclude that too many steps are being taken in these solutions for them to be optimal. In edition, a vast amount of nodes are being expanded. We believe that these issues are a cause of our algorithm's incapability to revisit goal nodes. Once an edge has been taken in the MST graph which is generated to represent the maze, it cannot be retaken. If two nodes are connected by more than one edge and are not directly adjacent, our implementation will not walk the multiple edges to from start to finish if they have already been visited. We hypothesise that adding a means of breaking down steps between non-adjacent nodes would fix this. We should note, however, that the path displayed in our solutions is what we believe to be the optimal path,

as it is the minimum spanning tree of the maze. It is just that these paths are being traversed in an incorrect, back-and-forth manner in which goals are visited in an improper order. The following tables organize the data used in making these conclusions:

	A*
Tiny Search	72
Small Search	244
Medium Search	402

Table 3. Multi-Goal A* Search Implementation; Step Cost

	A*
Tiny Search	90
Small Search	594
Medium Search	912

Table 4. Multi-Goal A* Search Implementation; Nodes Expanded

Conclusions

We were asked to implement breadth-first search, depth-first search, greedy-first search, and the A* search to solve a series of maze and search puzzles. While errors may be present in our A* search for multiple goals, the results of our first four search implementations is enough to allow us to analyse the optimality of these search implementations with respect to one another. DFS produces a suboptimal result quickly while BFS and A* produce optimal results, A* producing a result at a quicker pace. Greedy-first search is suboptimal, however, with the right heuristic and the right circumstance, it can perform with close comparison to BFS. In summary, the optimality of the path generated by a search algorithm varies with the implementation used; some searches are more efficient than others.